blackfalcon / **git-feature-workflow.md**

Last active a day ago • Report abuse

Git basics - a general workflow

⟨⟩ `git-feature-workflow.md`

There are many Git workflows out there, I heavily suggest also reading the atlassian.com Git Workflow article as there is more detail then presented here.

The two prevailing workflows are Gitflow and feature branches. IMHO, being more of a subscriber to continuous integration, I feel that the feature branch workflow is better suited.

When using Bash in the command line, it leaves a bit to be desired when it comes to awareness of state. I would suggest following these instructions on setting up GIT Bash autocompletion.

Basic branching

When working with a centralized workflow the concepts are simple, `master` represented the official history and is always deployable. With each now scope of work, aka feature, the developer is to create a new branch. For clarity, make sure to use descriptive names like `transaction-fail-message` or `github-oauth` for your branches.

#Protip: Although you may have a feature like 'user login and registration`, this is not considered appropriate to create a feature branch at this level, there is too much work to be done. It is better to break these large deliverables down to smaller bits of work that can be continuously integrated into the project. Remember, commit early and often.

Before you create a branch, be sure you have all the upstream changes from the `origin/master` branch.

Make sure you are on master

Before I pull, I make sure I am on the right branch. I have GIT Bash autocompletion installed, this tells me the branch in the prompt. Otherwise, the following command is good to know to list out the branches I have locally as well designate which branch I am currently on.

```
$ git branch
```

The checked out branch will have a `*` before the name. If the return designates anything other then `master` then switch to master

```
$ git checkout master
```

Once on master and ready to pull updates, I use the following:

```
$ git pull origin master
```

The `git pull` command combines two other commands, `git fetch` and `git merge`. When doing a `fetch` the resulting commits are stored as remote branch allowing you to review the changes before merging. Merging on the other hand can involve additional steps and flags in the command, but more on that later. For now, I'll stick with `git pull`.

Now that I am all up to date with the remote repo, I'll create a branch. For efficiency, I will use the following:

```
$ git checkout -b my-new-feature-branch
```

This command will create a new branch from `master` as well checkout out that new branch at the same time. Doing a `git branch` will list out the branches in my local repo and place a `*` before the branch that is checked out.

```
  master
* my-new-feature-branch
```

Do you have to be on master to branch from master?

No. There is a command that that allows me to create a new branch from any other branch while having checked out yet another branch. WAT?!

```
$ git checkout -b transaction-fail-message master
```

In that example, say I was in branch `github-oauth` and I needed to create a new branch and then checkout the new branch? By adding `master` at the end of that command, Git will create a new branch from master and then move me (checkout) to that new branch.

This is a nice command, but make sure you understand what you are doing before you do this. Creating bad branches can cause a real headache when trying to merge back into master.

Branch management

As I am working on my new feature branch, it is a good idea to commit often. This allows me to move forward without fear that if something goes wrong, or you have to back out for some reason, I don't lose too much work. Think of committing like that save button habit you have so well programed into you.

Each commit also tells a little bit about what I just worked on. That's important when other devs on the team are reviewing my code. It's better to have more commits with messages that explain the step versus one large commit that glosses over important details.

Commit your code

As I am creating changes in my project, these are all unseated updates. With each commit there most likely will be additions, and there will also be deletions from time to time. To get a baring of the updates I have made, lets get the status.

```
$ git status
```

This command will give you a list of all the updated, added and deleted files.

To add files, I can add them individually or I can add all at once. From the root of the project I can use:

```
$ git add .
```

In order to remove deleted files from the version control, I can again either remove individually or from the root address them all like so:

```
$ git add -u
```

I'm lazy, I don't want to think, so the following command I make heavy use of to address all additions and deletions.

```
$ git add --all
```

All the preceding commands will stage the updates for commitment. If I run a `git status` at this point, I will see my updates presented differently, typically under the heading of `Changes to be committed:` . At this point, the changes are only staged and not yet committed to the branch. To commit, do the following:

```
$ git commit -m "a commit message in the present tense"
```

It is considered best to illustrate your comment in the tense that this will do something to the code. It didn't do something in the past and it won't do something in the future. The commit is doing something now.

A bad example would be:

```
$ git commit -m "fixed bug with login feature"
```

A good example would be:

```
$ git commit -m "update app config to address login bug"
```

Comments are cheap. For more on how to write expressive commit messages, read 5 Useful Tips For A Better Commit Message.

Push your branch

When working with feature branches on a team, it is typically not appropriate to merge your own code into master. Although this is up to your team as how to manage these expectations, but the norm is to make use of pull requests. Pull requests require that you push your branch to the remote repo.

To push the new feature branch to the remote repo, simply do the following:

```
$ git push origin my-new-feature-brach
```

As far as Git is concerned, there is no real difference between `master` and a feature branch. So, all the same Git features apply.

My branch was rejected?

This is a special case when working on a team and the branch I am are pushing is out of sync with the remote. To address this, it's simple, pull the latest changes:

```
$ git pull origin my-new-feature-branch
```

This will fetch and merge any changes on the remote repo into my local brach with the changes, thus now allowing you to push.

Working on remote feature branches

When I am are creating the feature branch, this is all pretty simple. But when I need to work on a co-workers branch, there are a few additional steps that I follow.

Tracking remote branches

My local `.git/` directory will of course manage all my local branches, but my local repo is not always aware of any remote branches. To see what knowledge my local branch has of the remote branch index, adding the `-r` flag to `git branch` will return a list.

```
$ git branch -r
```

To keep my local repo 100% in sync with deleted remote branches, I make use of this command:

```
$ git fetch -p
```

The `-p` or `--prune` flag, after fetching, will remove any remote-tracking branches which no longer exist.

Switching to a new remote feature branch

Doing a `git pull` or `git fetch` will update my local repo's index of remote branches. As long as co-workers have pushed their branch, my local repo will have knowledge of that feature branch. By doing a `git branch` you will see a list of my local branches. By doing a `git branch -r` I will see a list of remote branches. There is a good chance that the new feature branch is not in my list of local branches.

The process of making this remote branch a local branch to work on is easy, simply checkout the branch.

```
$ git checkout new-remote-feature-branch
```

This command will pull it's knowledge of the remote branch and create a local instance for me to work on.

The Pull request

The pull request is where the rubber meets the road. As stated previously, one of the key points of the feature branch workflow is that the developer who wrote the code does not merge the code with master until there has been a peer review. Leveraging Github's pull request features, once you have completed the feature branch and pushed it to the repo, there will be an option to review the diff and create a pull request.

In essence, a pull request is a notification of the new code in an experience that allows a peer developer to review the individual updates within context of the update. For example, if the update was on line 18 of `header.haml`, then you will only see `header.haml` and a few lines before and after line 18.

This experience also allows the peer reviewer to place a comment on any line within the update. This will be communicated back to the editor of origin. This review experience really allows for everyone on the team to be actively involved in each update.

Once the reviewer has approved the editors updates, there are two ways to merge in the code. One from the Github interface and another from the command line.

Merging the code

Although I can merge from Github's interface, it is preferred to do it from the command line. After all, what happens if Github merging tools are broken? It can happen.

So let's assume that I created a feature branch, edited code and pushed it to the repo. I initiated a pull request and the code update was approved. Here are the steps I will go through to merge the new code. Keep in mind, this illustration is simply managing code, this is not including running tests, while it can be worked into this workflow, that is a separate concern.

1. Make sure that I have the latest version of the feature branch from the remote repo

```
$ git checkout my-feature-branch
$ git pull origin my-feature-branch
```

2. Make sure that the feature branch is up to date with `master`, while in the feature branch, execrate the following:

```
$ git pull origin master
```

If there are any conflicts, best to address them here.

3. Now that I know that the feature branch is up to date with the remote repo and that it has the latest code from `master`, I can now merge these branches. I also need to make sure that my local `master` branch is up to date as well.

```
$ git checkout master
$ git pull origin master
$ git merge --no-ff my-feature-branch
```

Notice the `--no-ff` flag in the merge command. This flag keeps the repo branching history from flattening out. If I were to look at the history of this branch, using GitX for example, when using the `--no-ff` flag, I will see the appropriate bump illustrating the history of the feature branch. This is helpful information. If I didn't use this flag, then Git will move the commit pointer forward.

I can either enter the `--no-ff` flag each time I merge or I can set this as my default. I prefer the default option. Running the following command will update the global gitconfig.

```
$ git config --global merge.ff false
```

And of course, setting the bit back to `true`, will return the default setting to fast forward with merging.

NOTE: there is a mild side-effect that will happen when you set this flag to `false`. Every time you do a `pull` this will not fast forward your local repo and basically make this a new commit. That's ok, but you will see this in the commit logs

```
Merge branch 'master' of github.com: ...
```

and

```
# Your branch is ahead of 'origin/master' by 1 commit.
```

If you are ok with this, then keep the flag. If this annoys or bothers you, do not use the flag and set `--no-ff` manually with each merge.

4. Now that I have merged the code, the feature branch by definition is obsolete. First, delete the branch from the local repo.

```
$ git branch -d my-feature-branch
```

The `-d` flag for delete will typically delete any branch. Remember, you can't delete a branch you have checked out.

If you happen to see the following error when deleting a branch, then simply replace the `-d` with `-D` to force the delete.

```
error: The branch 'name-of-branch' is not an ancestor of your current HEAD.
```

5. If the feature branch was pushed to the repo, as it should have been per the workflow we described, you will want to delete this from the remote repo as well.

```
$ git push origin --delete my-feature-branch
```

Shortcuts using aliases

There are some steps in there that we should just be doing all the time. What about making a single command alias that will cycle through all these commands just so we know things are always in good shape? Yup, we can do that.

In Bash

Using Git and Bash is like using a zipper and pants. They just go together. Creating a Bash alias is extremely simple. From your Terminal, enter

```
$ open ~/.bash_profile
```

This will open a hidden file in a default text editor. If you have a shortcut command for opening in an editor like Sublime Text, use that to open the file. In the file add the following:

```
alias refresh="git checkout master && dskil && git pull && git fetch -p"
```

The alias `dskil` is useful for removing annoying `.DS_Store` files. You should have a `.gitignore` file that keeps these out of version control, but I like to keep a clean house too. To make that work, add the following:

```
alias dskil="find . -name '*.DS_Store' -type f -delete"
```

With this in your `.bash_profile`, you simply need to enter `refresh` in the command line and POW!

In Powershell

If you are on Windows and using Powershell, you can use the same aliases, but the set up is different. The article Windows PowerShell Aliases is a good tutorial of the process.

Summary

Following this simple workflow has helped keep my projects clean and `Git hell` free. I hope it serves you well too.

But ... the story doesn't end here. Well, the post does, but the story does not. There are many more advanced features that your team can use. I am sure some of you reading this will say "What about rebasing?" This is a perfectly practical addition to this workflow and many teams use it. But out of all the teams I have ever worked for, only one has ever made use of this feature. Just saying.

Outside of that, for more in-depth learning on Git, I invite you to read the Git book, it's free and contains awesome learning.

---

aao13 commented on Feb 22, 2016

typical Linux warped thinking - simplest of all source control operation , such as merge 2 versions of the same file , becomes convoluted to the extreme, they way the author savors the experience is asinine

komaxx commented on Mar 2, 2016

Good, concise description. Thank you!

BroVic commented on Apr 18, 2016

The article was helpful. Thanks!

CoreyGumbs commented on Oct 23, 2016

Thank you so much.

2gingersnaps commented on Oct 30, 2016

Bravo!

upendrak commented on Feb 22, 2017

Nice tutorial. Thanks

kimstone commented on Mar 20, 2017

Very well written. Great breakdown.

nitchow commented on May 16, 2017

Superb article - this is pretty much my ref when I do pull requests.

dreftymac commented on Aug 18, 2017

I wish to react with a thumbs-up emoji.

NoashX commented on Aug 28, 2017 • edited ▾

"Make sure that the feature branch is up to date with master, while in the feature branch, `execrate` the following:"

I think you mean execute?

sacredabhishek commented on Oct 27, 2017

helpful article **@blackfalcon**

lifeF commented on Nov 2, 2017

thanks

stanlee321 commented on Nov 9, 2017

Gracias!

boundaries-io commented on Nov 9, 2017

Thank you sir.

fabiomarcos70 commented on Nov 25, 2017

Very smart and helpfull article. Thanks a lot

comtechnet commented on Dec 13, 2017

I think there is a typo ...

you wrote ...

Basic branching

When working with a centralized workflow the concepts are simple, master represented the official history and is always deployable. With each now scope of work, aka feature, the developer is to create a new branch. For clarity, make sure to use descriptive names like transaction-fail-message or github-oauth for your branches.

I think you mean ....

Basic branching

When working with a centralized workflow the concepts are simple, master represented the official history and is always deployable. With each **new** scope of work, aka feature, the developer is to create a new branch. For clarity, make sure to use descriptive names like transaction-fail-message or github-oauth for your branches.

SkybuckFlying commented on Dec 25, 2017 • edited ▾

Deleting anything is a big fat no-no. Certainly don't like the "delete branch concept". If for example remote server is nuked, bye bye any changes.
Deleting changes is just dumb ? Why the hell would anybody think this is a good idea is beyond me ? These branches offer a nice insight into what changes were made... Deleting that is just dumb dumb dumb dumb dumb.
At least some history is kept in the commits/merges to see what was changed ?! But what if merger decides to merge many branches before commiting ?! Again these nice smaller litte branches of changes would be lost ! Very odd idea to delete anything for a version control system. Terrabytes plenty...

A better idea would be to "close a branch" and marking it as "merged". And that's coming from a total git noob=me. I think tagging could be used for this... not sure.

Ok now that I worked with git myself a bit I see how easy it is to make a mistake... typos... and being at the wrong head... so now I understand this feature... it's still all pretty dangerous though ! =D

tomcoders commented on Jan 26

Thank you for this concise git user guide.

doha12 commented on Jan 31

Using git fetch upstream master pulled in the changes from the master branch on the upstream remote repository.

What single command would we use if we want to fetch the upstream/master changes and merge them into the master branch?

suessflorian commented on Feb 12

Cheers

AbdelrahmanKabel commented on Feb 20

@doha12
"git pull " is the same thing as a "git fetch" + "git merge"
so the single command is:
"git pull upstream master"

Omar-Suliman commented on Feb 23

thank you so much

Jyothireddyb commented on Mar 12

I couldnot get what a pullrequest is???

Vanshika-Mittal commented on May 28

How do I tie my GitBash to my GitHub account? Also, can you tell me where my GitBash commits go if its not tied to my account.

eylulyurdakul commented on Oct 10

This is a super useful article with very clear explanations, thank you!

rtsuh commented 27 days ago

Very helpful - thank you!

kushalmahajan commented a day ago

You just documented, what I would have documented myself. You haven't missed a point. In fact learnt a couple of things more. --no-ff and alias features.